실시간 내장형 시스템의 설계를 위한 비선점형 고정우선순위 스케쥴링
(Non-Preemptive Fixed Priority Scheduling for Design of Real-Time Embedded Systems)

박 문 주 [†]
(Moonju Park)

요 약 유비쿼터스 환경에서 널리 사용되고 있는 내장형 시스템에서는 메모리 사용량의 감소와 신뢰성 등의 이유로 쓰레드 기반 프로그래밍 모델보다는 이벤트-구동형 프로그래밍 모델을 채용하는 경우가 많다. 그러나 내장형 시스템의 소프트웨어가 점점 더 복잡해지면서, 내장형 시스템을 위한 소프트웨어를 이벤트-구동형 프로그래밍 모델의 단일 이벤트 핸들러로 프로그램 하는 것은 매우 어려운 과제가 되고 있다. 본 논문에서는 내장형 시스템의 설계에 비선점형 스케쥴링 기법을 도입하기 위하여, 다항시간의 스케쥴 가능성 평가를 위한 충분조건과 이를 이용한 효율적인 스케쥴 가능성 검사 방법을 제시하며, 또한 내장형 시스템에서의 서브-태스크의 활용이 비선점형 스케쥴링을 채용했을 때의 단점인 낮은 처리기 이용률을 극복할 수 있는 방안임을 보인다.

키워드 : 내장형 시스템, 실시간, 스케쥴링

Abstract Embedded systems widely used in ubiquitous environments usually employ an event-driven programming model instead of thread-based programming model in order to create a more robust system that uses less memory. However, as the software for embedded systems becomes more complex, it becomes hard to program as a single event handler using the event-driven programming model. This paper discusses the implementation of non-preemptive real-time scheduling theory for the design of embedded systems. To this end, we present an efficient schedulability test method for a given non-preemptive task set using a sufficient condition. This paper also shows that the notion of sub-tasks in embedded systems can overcome the problem of low utilization that is a main drawback of non-preemptive scheduling.

Key words : embedded system, real-time, scheduling

1. Introduction

Networked embedded systems such as wireless sensor nodes or low-end communication devices are usually designed to be event-driven, so they are reactive and power efficient. One of the main drawbacks in use of the preemptive thread-based programming model for such systems is the unnecessary concurrency that is introduced by the multi-thread model that forces programmers to cope with synchronization between threads, consequently makes programs not robust [1]. Another problem with the preemptive thread-based programming model is the stack memory overhead that is especially burdensome in embedded systems. However, when a system requires run-time scheduling, event-driven programming based on control flow state machines is difficult.

Consider an example of digital signal processing in [2]. An embedded system includes a function

· 이 논문은 인천대학교 2007년도 자체연구비 지원에 의하여 연구되었음

† 정 회 원 : 인천대학교 컴퓨터공학과 교수
mpark@incheon.ac.kr

논문접수 : 2008년 1월 2일
심사완료 : 2008년 11월 17일

that converts CD sampling rate (44.1 kHz) to digital audio tape (DAT) rate 48 kHz. First, a polyphase FIR filter raises the sampling rate from 44.1 kHz to 88.2 kHz. The next stage raises the sampling rate to 117.6 kHz. Then the sample-and-hold interface may duplicate or drop samples depending on whether the sampling clock for the DAT is faster or slower than 48 kHz. The discontinuities introduced during re-sampling procedure are smoothed out by anti-aliasing filters. Because the relative rates of the clocks in this example are not known off-line, it is impossible to statically determine the execution order of the entire system. Thus this kind of systems requires run-time scheduling.

Non-preemptive scheduling on uniprocessor systems can eliminate the synchronization overhead of the resource-protecting mechanism. For example, [2] showed that non-preemptive fixed priority scheduling can be used for real-time signal processing applications because the amount of processor state information (including the stack to be stored) can be reduced. Since we can expect much lower overhead and smaller memory requirements at run time with non-preemptive scheduling, it can overcome the drawbacks of the thread-based programming model. However, the processor utilization of non-preemptive scheduling can be arbitrarily small [3] if tasks are not carefully designed. Also, to design a system, we should determine the crucial system parameters like how often the task should run and how long the task can occupy the processor.

In this paper, we find a sufficient condition for schedulability with non-preemptive fixed-priority scheduling to efficiently check the schedulability, and we show that the proposed test is a good approximation of exact schedulability by simulation. While current polynomial time schedulability tests are applied only to RM priority ordering, the proposed method can be applied to any priority ordering. To attack the problem of low utilization of non-preemptive scheduling, it is shown that the notion of sub-tasks in embedded systems can enhance the processor utilization in non-preemptive fixed priority scheduling. We present a method to calculate the worst case response time of tasks when tasks are decomposed into sub-tasks.

The rest of the paper is organized as follows. Section 2 gives a review of related work. Section 3 summarizes the assumptions and terms that are used in this paper. In section 4, we present an efficient schedulability test method for a given task set. In section 5, we tackle the problem of low processor utilization, which is the major problem with non-preemptive scheduling. Finally, section 6 concludes our work.

2. Related Work

Though non-preemptive schedulers have received less attention than preemptive ones, some results are known for dynamic priority non-preemptive scheduling. It is known that the general problem of finding a feasible schedule in an idling non-preemptive context is NP-complete. Jeffay et al. [3] showed that Earliest Deadline First (EDF) is optimal among non-idling schedulers, and non-preemptive scheduling of concrete periodic tasks is NP-hard in the strong sense. They also found a necessary and sufficient condition for schedulability under non-idling EDF. It was shown in [4] that the worst case response time can be calculated using the results of [5].

While EDF is optimal for both preemptive and non-preemptive scheduling, it was shown in [4] that Deadline Monotonic (DM) scheduling algorithm, which is optimal for preemptive fixed priority scheduling, is not optimal for non-preemptive fixed priority scheduling, so Rate Monotonic (RM) scheduling is not optimal either. In [4], they found a method to obtain the worst case response time of tasks, but the calculation of the worst case response time takes pseudo-polynomial time. Based on the results of [4], [6] proposed fixed-priority scheduling with a preemption threshold that subsumes both preemptive and non-preemptive scheduling.

Though we can determine whether a given task set is schedulable, we do not have a simple and efficient schedulability test method like the utilization bound of preemptive RM in [7]. Other scheduling methods for designing embedded real-time systems are based on off-line scheduling analysis. However, making static schedules is NP problem even when tasks have fixed priorities [8]. One of the recent research results on off-line scheduling can be found in Pre-scheduling [9]. In Pre-scheduling, task schedules are generated off-line based on EDF using Linear Programming, and scheduled according to the off-line schedule with sporadic tasks on-line. By off-line analysis, we can validate schedules of the system design with given parameters. But considering the software is modified very often in development process, the high complexity of off-line algorithms may affect the development time.

To make non-preemptive scheduling useful in embedded systems like digital signal processing units, the problem of low processor utilization must be solved. To cope with the problem, techniques for decomposing a task into several pieces have been used as in [9] and [10]. But most of the research has been done with EDF-based schemes. Another approach to the problem can be found in [11], which suggest the use of (m,k)-firm deadline model [12] to drop some task instances, and this approach also is based on EDF scheduling scheme.

3. System Model and Assumptions

A periodic task is denoted by τ_i. A periodic task set is represented by the collection of periodic tasks, $\tau=\{\tau_i\}$. Each τ_i is a pair (T_i, C_i) where T_i is the period and C_i is the worst case execution time, and $T_i \ge C_i > 0$. The relative deadline of a task is the same as its period. This requires that if τ_i is invoked at time t_x, τ_i must have C_i units of processor time allocated to it in the interval $[t_x, t_x + T_i]$.

A concrete task has a specified release time, or the time of the first invocation. The difficulty of scheduling tasks can be affected by the release time [3]. A periodic task set is said to be schedulable if and only if all of the concrete task sets that can be generated from the periodic task set are schedulable. We will consider only periodic task sets in this paper.

The following is a summary of the assumptions that are used in this paper.

- There is only one processor.
- Scheduling overhead can be ignored.
- There is no release jitter. (i.e., tasks become ready when they arrive.)
- Time is represented by an integer number, so time is discrete and the clock ticks are indexed by integer numbers, as in [3].

When scheduling overhead cannot be ignored, the overhead can be counted as a part of the worst case execution time of tasks. For embedded systems, we can use many currently available techniques for accounting the worst case execution time of tasks [13].

Though we assume that time and all task parameters are integer, this assumption does not affect the generality of the schedulability results since it was shown that a common continuous schedule exists if and only if a feasible discrete schedule exists [14]. Therefore, the integral time assumption ensures the generality of this paper's results.

The assumptions used in this paper are well accepted in real-time literature. The same assumptions have been used for analyzing the Controller Area Network (CAN), and the analysis method is used successfully as a basis of a commercial CAN schedulability analysis tool by companies like Volvo Communications Technologies AB, in design and development of the CAN and electronics systems for their vehicles [15].

For a given task τ_i, we define $hp(\tau_i)$ as the subset of τ that consists of tasks with priority higher than τ_i. On the other hand, $lp(\tau_i)$ is the set of tasks with lower priority than τ_i. For the convenience of discussion, without any loss of generality, we assume that tasks are sorted in non-decreasing order by period. That is, for any pair of tasks τ_i and τ_j, if $j < i$ then $T_j \le T_i$.

4. Schedulability Test Methods for Non-Preemptive Scheduling

The first step in designing a system is to determine whether or not tasks can be successfully scheduled with given system parameters (resources). In this section, we will present conditions for successful non-preemptive scheduling.

The following lemma from [4] shows that the concept of level-i busy period [16] is also useful in determining the schedulability of non-preemptive systems.

Lemma 1. A periodic task τ_i has the largest response time in a level-i busy period that is obtained by releasing all of the tasks τ_j such that $\tau_j \in hp(\tau_i)$ and τ_i simultaneously at time $t=0$, and the task τ_k such that $\tau_k \in lp(\tau_i)$ and $C_k = \max\{C_l | l > i\}$ at time $t=-1$.

The level-i busy period given in Lemma 1 is basically similar to the critical interval of the dynamic priority driven scheduler in [3]. The interval given in Lemma 1 is an extension of the concept of the critical instant in [7]. Using Lemma 1, [4] found that the worst case response time of a task τ_i is given by the following theorem.

Theorem 1. The worst case response time of any task τ_i is given by

$$r_i = \max\{w_{i,q} + C_i - qT_i | q = 0, 1, \ldots, Q_i\}$$

where

$$w_{i,q} = qC_i + \sum_{\tau_j \in hp\{\tau_i\}} \left(1 + \left\lfloor \frac{w_{i,q}}{T_j} \right\rfloor\right) C_j + \max\{C_k - 1 | \tau_k \in lp(\tau_i)\}$$

and $Q_i = \left\lfloor \frac{L_i}{T_i} \right\rfloor$, and where L_i is the length of the longest level-i busy period in non-preemptive context.

Theorem 1 gives us an exact schedulability test method for non-preemptive fixed priority scheduling: it is trivial that a task set is schedulable if and only if $r_i \le T_i$ for $\forall \tau_i \in \tau$. Though we can determine the schedulability exactly using Theorem 1, the time complexity of the test is high because it runs in pseudo-polynomial time. Using the results of [7] and based on the Liu and Layland's utilization bound (LL-bound), we can use the following sufficient condition as a schedulability test:

A task set τ is schedulable by RM if

$$\forall \tau_i \in \tau,\ \sum_{j=1}^{i-1} \frac{C_j}{T_j} + \frac{C_i + B_i}{T_i} \le i(2^{\frac{1}{i}} - 1)$$

where B_i is the worst case blocking time of τ_i.

By considering the processor as a shared resource, we can determine the schedulability of the task set with $B_i = \max\{C_k | k > i\}$. However it was shown in [16] that LL-bound is pessimistic.

To get less pessimistic utilization bound, [7] suggested a schedulability test using the exact schedulability test method in [16] with Priority Ceiling Protocol (PCP) [17] by considering the processor as a shared resource as follows:

A task set τ is schedulable by RM if

$$\forall \tau_i \in \tau,\ \min_{(k,l) \in R_i} \sum_{j=1}^{i-1} \frac{C_j}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil + \frac{C_i}{lT_k} + \frac{B_i}{lT_k} \le 1$$

where B_i is the worst case blocking time of τ_i and

$$R_i = \{(k,l) | 1 \le k \le i, 1 \le l \le \lfloor T_i / T_k \rfloor\}.$$

However, the test still runs in pseudo-polynomial time, though it provides only a sufficient condition. Also, the above test methods can be applied only to RM scheduling.

To find an efficient schedulability test method that runs in polynomial time and that is less pessimistic, we first introduce the following function:

$$G_i(t) = \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{t}{T_j} \right\rceil C_j.$$

The function $G_i(t)$ represents the amount of execution time that is requested by tasks with higher priority than τ_i, from time zero to time t when these tasks are released at time 0.

Lemma 2. For a task τ_i, the maximum interference due to τ_j with higher priority than τ_i when τ_j and τ_i are simultaneously released at time 0 is less than or equal to:

$$\left\lceil \frac{T_i}{T_j} \right\rceil C_j,\ \text{if}\ G_i(t)\left(\left\lfloor \frac{T_i}{T_j} \right\rfloor T_j\right) + C^i_{\max} - 1 \ge \left\lfloor \frac{T_i}{T_j} \right\rfloor T_j$$
$$\left\lfloor \frac{T_i}{T_j} \right\rfloor C_j,\ \text{otherwise}$$

where $C^i_{\max}$ is the maximum worst case execution time among tasks in $lp(\tau_i)$.

Proof:

The last activation of the task τ_j before T_i is made at time $T_i - (T_i \bmod T_j)$, which is equal to $\left\lfloor \frac{T_i}{T_j} \right\rfloor T_j$. Let this value be L_{ij}. By Lemma 1, the largest response time is obtained by releasing all tasks in $lp(\tau_i)$ with task τ_i simultaneously and the task with $C^i_{\max}$ at time $t=-1$. Thus the maximum interference in the execution of τ_i is less than or equal to $G_i(t)\left(\left\lfloor \frac{T_i}{T_j} \right\rfloor T_j\right) + C^i_{\max} - 1$.

If the total execution time requested from $hp(\tau_i)$ before L_{ij} plus $C^i_{\max} - 1$ is greater than or equal to L_{ij}, τ_i might not run before L_{ij} because the processor should serve the higher priority tasks and the lower priority task that is released at time $t = -1$. In this case, the last activation of τ_j could be executed prior to τ_i, but if $T_i \bmod T_j = 0$, the last activation of τ_j will interfere with the next activation of τ_i. Thus the maximum interference due to τ_j cannot exceed $\left\lceil \frac{T_i}{T_j} \right\rceil C_j$.

When the total execution time that is requested from $hp(\tau_i)$ before L_{ij} plus $C^i_{\max} - 1$ is less than L_{ij}, τ_i should have a chance to be scheduled before L_{ij}. Since tasks are non-preemptive, once task τ_i was executed it runs until it finishes its job, so the last activation of τ_j does not interfere with τ_i. Thus the maximum interference due to τ_j is given by $\left\lfloor \frac{T_i}{T_j} \right\rfloor C_j$. ■

Using Lemma 2, a sufficient condition for successful non-preemptive RM scheduling can be derived as the following theorem.

Theorem 2. A periodic task set τ is schedulable if

$$C^i_{\max} - 1 + C_i + \sum_{j=1}^{i-1} I_{ij} \le T_i \text{ for } \forall \tau_i \in \tau \text{ where}$$

$$C^i_{\max} = \begin{cases} \max\{C_k | k > i\}, & \text{for } i < n \\ 1, & \text{if } i = n \end{cases}$$

and

$$I_{ij} = \begin{cases} \left\lceil \frac{T_i}{T_j} \right\rceil C_j, \text{ if } G_i\left(\left\lfloor \frac{T_i}{T_j} \right\rfloor T_j\right) + C^i_{\max} - 1 \ge \left\lfloor \frac{T_i}{T_j} \right\rfloor T_j \\ \left\lfloor \frac{T_i}{T_j} \right\rfloor C_j, otherwise \end{cases}$$

Proof:

By Lemma 2, the maximum interference due to tasks with higher priority than τ_i is less than or equal to $\sum_{j=1}^{i-1} I_{ij}$. By Lemma 1, the maximum interference due to tasks with lower priority than τ_i is given by $C^i_{\max} - 1$. Therefore if $C^i_{\max} - 1 + C_i + \sum_{j=1}^{i-1} I_{ij} \le T_i$, τ_i is schedulable. ■

As an example of application of Theorem 2, let us consider a feedback control system wherein the control loops are closed through real-time network. This kind of system is called Networked Control System (NCS) [18]. Let us take an example in [18], where there are 3 NCSs and the transmission time of the NCSs is 4 ms, using DeviceNet specification. They calculated the transmission period of NCSs as: 14.6 ms, 15.0 ms, and 16.8 ms using LL bound. For a task set with 3 tasks, LL bound requires utilization is no more than $3(2^{\frac{1}{3}} - 1) \approx 0.7798$. Since (4/14.6+4/15.0+4/16.8) = 0.7787, the periods they calculated can be thought as tight when LL bound is considered. However, applying Theorem 2, we can obtain shorter periods for NCSs.

To apply Theorem 2, let us change the periods to $146t$, $150t$, and $168t$, and transmission time to $40t$ where t is 0.1 ms. Since the possible shortest period is $100t$, let us change the Task 1's period to $100t$. Then we have

Task 1: $(40t - 1t) + 40t = 79t < 100t$

Thus Task 1 is schedulable with the period of 10 ms. And since $G_2(100t) + 40t - 1t = 79t < 100t$, Task 2's period can be as short as $120t$, that is, 12 ms.

Task 2: $(40t - 1t) + 40t + 40t = 119t < 120t$

Finally, since $G_3(100t) = 80t < 100t$ and $G_3(120t) = 120t \ge 120t$, Task 3's period can be as short as 160 t, that is, 16 ms.

Task 3: $(1t - 1t) + 40t + 40t + 2 \times 40t = 160t$

Therefore, by applying Theorem 2, we can obtain tighter periods for NCSs in [18], 10 ms, 12 ms, 16 ms. As shown in this example, the proposed method can be efficiently used to design reactive embedded systems to increase the system utilization. As a result of the enhancement, the system utilization increases to 0.9833.

To see the effectiveness of the proposed test method, we conducted a simulation by using the randomly generated tasks. Task parameters were generated using the UNIX random() function. Periods are smaller than 100,000 and the worst case execution times were generated to be smaller than 10,000. Any one task does not have a utilization of over 70%, and all of the tasks have at least 0.5% utilization. A total of 1,300 task sets, containing 8,337 tasks, were generated and tested. We compared the percentage of task sets that were determined to be schedulable by preemptive and non-preemptive RM scheduling. The results are summarized in Table 1. For comparison, the percentage of preemptive task sets that were determined to be schedulable by preemptive RM is presented also. The schedulability of preemptive RM is calculated using the method in [16].

As shown in Table 1, the schedulability test given in Theorem 2 performs better than the test that was based on LL-bound and PCP. For task sets with a processor utilization of lower than 80%, the proposed test method determines the schedulability of all of the task sets exactly. For task sets with 80% of the average utilization, the proposed method shows only 4% of the error in determining the schedulability (4% of task the sets are determined as "not schedulable," though they are, in fact, schedulable). For task sets with high utilization, the error in determining the schedulability becomes larger, but it shows a better ratio than the PCP-based method shows. Furthermore, the test based on Theorem 2 runs in polynomial time while the PCP-based test runs in pseudo-polynomial time. Thus, the proposed method runs faster than the PCP-based test and it has less error in determining the schedulability of the task sets.

5. Enhancement of Processor Utilization for Embedded Systems

The major drawback in programming an event-driven system using non-preemptive scheduling is that the processor utilization can be very small in order to guarantee the deadlines of tasks. In practical situations, however, we can achieve higher utilization by carefully choosing the task parameters. For example, in preemptive scheduling, harmonic task sets [19] (each task's period is an exact divisor of each longer period) and load scaling method [20] can achieve higher utilization. In this section, we show that the notion of sub-tasks in embedded systems can enhance the processor utilization.

Load scaling by increasing the period that is effective in preemptive scheduling often does not work for non-preemptive scheduling. Let us take a task set {(20,10), (30,15)} for example. This task set is not schedulable by preemptive RM, but if we adjust the task parameters to be {(20,10), (40,20)} (i.e., make the period harmonic but keep the task utilization the same as before), then the task set becomes schedulable by preemptive RM. But in non-preemptive scheduling, because lengthening the period also makes the worst case execution time of the lower priority task larger, the task set still remains unschedulable by non-preemptive RM.

From Theorem 2, we notice that if we can reduce the interference that is caused by the lower priority

Table 1 Schedulability test results

Average utilization	Preemptive RM	Non-Preemptive RM			
		Exact test	Theorem 2	PCP test	LL test
10%	100%	100%	100%	100%	100%
20%	100%	98%	98%	98%	98%
30%	100%	96%	96%	96%	96%
40%	100%	92%	92%	92%	92%
50%	100%	90%	90%	90%	90%
60%	100%	93%	93%	93%	92%
70%	100%	78%	78%	75%	73%
80%	96%	74%	71%	65%	0%
90%	46%	34%	19%	13%	0%

task, the processor utilization may increase. From this investigation, we see that splitting tasks may enhance processor utilization. In many embedded applications, tasks consist of sub-tasks, where a sub-task is a part of the job a task should do at each period [15]. This situation can be modeled with task τ_i and with sub-tasks $\{\tau_{i,1}, \tau_{i,2}, \cdots, \tau_{i,N_i}\}$, where N_i is the number of sub-tasks of τ_i. Every $\tau_{i,j}$ has the same period T_i and priority, but it has its own worst case execution time $C_{i,j}$. Note that $\sum_{j=1}^{N_i} C_{i,j} = C_i$.

The next theorem shows that by splitting a task into small pieces, we can enhance the processor utilization.

Theorem 3. For a given task set $\tau = \{\tau_i\}$ where each task τ_i has sub-tasks $\{\tau_{i,1}, \tau_{i,2}, \cdots, \tau_{i,N_i}\}$, the blocking time experienced by a task τ_i is given by

$$\max\{C_{k,j} | i < k, 1 \le j \le N_k\} - 1.$$

Proof:

Let t_1 be the release time of τ_i's instance and t_2 be the completion time of the instance. Note that there is no idle time in time interval $[t_1, t_2]$. Since τ_i's instance completes its execution at time t_2, only jobs of tasks in $hp(\tau_i)$ and τ_i can execute in the time interval $[t_1, t_2]$ among tasks which release their instances in the time interval. By Lemma 1, the largest response time is obtained by releasing tasks in $hp(\tau_i)$ with τ_i simultaneously. Let us assume that a task $\tau_k \in lp(\tau_i)$ is released at time $t_1 - 1$. Then all of the sub-tasks of $\tau_k = \{\tau_{k,1}, \tau_{k,2}, \cdots, \tau_{k,N_k}\}$ become ready at $t_1 - 1$ but only one sub-task can start execution in a uniprocessor system. Because of the non-preemptability, the sub-task runs to completion, thus making the response time of τ_i longer by $\max\{C_{k,j}\} - 1$ at most. ■

The worst case response time of tasks are also affected by the sub-tasking operation. The next theorem shows the worst case response time of tasks with sub-tasks can be calculated with the same time complexity as Theorem 1.

Theorem 4. For a given task set $\tau = \{\tau_i\}$ where each task τ_i has sub-tasks $\{\tau_{i,1}, \tau_{i,2}, \cdots, \tau_{i,N_i}\}$, the worst case response time of any task τ_i is given by

$$r_i = \max\{w_{i,q} + C_{i,N_i} - qT_i | q = 0, 1, \ldots, Q_i\}$$

where

$$w_{i,q} = qC_i + \sum_{\tau_j \in hp\{\tau_i\}} \left(1 + \left\lfloor \frac{w_{i,q}}{T_j} \right\rfloor\right) C_j + C_i - C_{i,N_i} + \max\{C_{k,j} | i < k, 1 \le j \le N_k\} - 1$$

and $Q_i = \left\lfloor \frac{L_i}{T_i} \right\rfloor$, and where L_i is the length of the longest level-i busy period in non-preemptive context.

Proof:

By Theorem 3, the maximum blocking time due to a lower priority task τ_k cannot exceed $\max\{C_{k,j}\}$. Thus, it is clear that the blocking time is $\max\{C_{k,j} | i < k, 1 \le j \le N_k\} - 1$. Let us denote this value as B_i.

Let $\tau_{i,x}$ be the last task which finishes its job when all of $\tau_{i,j}$ are simultaneously released. Then, the worst case response time of τ_i is the same as the worst case response time of the sub-task $\tau_{i,x}$. Without any loss of generality, we can assume that $\tau_{i,x} = \tau_{i,N_i}$. By definition, τ_{i,N_i} has the worst case execution time of C_{i,N_i}.

By Theorem 1, the worst case response time of τ_{i,N_i} is given by

$$\begin{aligned}
w_{iN_i,q} &= qC_{i,N_i} + \sum_{\tau_{j,k} \in hp\{\tau_{i,N_i}\}} \left(1 + \left\lfloor \frac{w_{iN_i,q}}{T_j} \right\rfloor\right) C_{j,k} + B_i \\
&= qC_{i,N_i} + \sum_{\tau_j \in hp\{\tau_i\}} \left(1 + \left\lfloor \frac{w_{iN_i,q}}{T_j} \right\rfloor\right) C_j + \sum_{k=1}^{N_i-1} \left(1 + \left\lfloor \frac{w_{iN_i,q}}{T_i} \right\rfloor\right) C_{i,k} + B_i \\
&= qC_{i,N_i} + \sum_{\tau_j \in hp\{\tau_i\}} \left(1 + \left\lfloor \frac{w_{iN_i,q}}{T_j} \right\rfloor\right) C_j + (1+q)(C_i - C_{i,N_i}) + B_i \\
&= C_i + \sum_{\tau_j \in hp\{\tau_i\}} \left(1 + \left\lfloor \frac{w_{iN_i,q}}{T_j} \right\rfloor\right) C_j + C_i - C_{i,N_i} + B_i.
\end{aligned}$$

By substituting $w_{iN_i,q}$ with $w_{i,q}$, the theorem is proved. ■

To see how the processor utilization is enhanced, we compare the percentage of schedulable task sets of preemptive RM scheduling and non-preemptive RM scheduling with sub-tasks using the task sets that were described in the previous section. In this experiment, when a task set is not schedulable by non-preemptive RM, we just divide the task with the maximum execution time into two sub-tasks such that each task has exactly half of the execu-

Table 2 Enhanced schedulability with task splitting

Average utilization	Preemptive RM	Non-Preemptive RM	Non-Preemptive RM with sub-tasks
10%	100%	100%	100%
20%	100%	98%	99%
30%	100%	96%	100%
40%	100%	92%	97%
50%	100%	90%	98%
60%	100%	93%	97%
70%	100%	78%	89%
80%	96%	74%	80%
90%	46%	34%	44%

tion time of the task. The results of the simulation are summarized in Table 2.

Even though we simply split a task into two, the non-preemptive RM scheduling with sub-tasks can successfully schedule most of the task sets when the load of the task sets is not heavy. And in overload situation (utilization as much as 90%), the percentage of schedulable tasks in non-preemptive scheduling is similar to that in preemptive scheduling.

6. Conclusion and future work

In this paper, we showed that event-driven embedded systems can be modeled as non-preemptive real-time systems. By employing a non-preemptive scheduling algorithm, we can achieve the advantages of an event-driven programming model and of thread-based programming model. The advantages include robustness, small memory usage by sharing a stack, no data races, simplicity, etc. To design a system, we should determine the crucial system parameters like how often the task should run and how long the task can occupy the processor. For this purpose, we have presented a sufficient condition for the existence of non-preemptive fixed priority scheduling that runs in polynomial time. Experimental results show that the proposed method is a good approximation of the exact schedulability test.

Though non-preemptive scheduling has many advantages, it suffers from low processor utilization. To attack the problem, we suggested enhancing the processor utilization by splitting a task into sub-tasks. The sub-tasks have the same period as the task, and the total amount of maximum execution time remains unchanged. By modeling each stage of the computational jobs in embedded systems with the same frequency as a sub-task, we can achieve higher utilization.

This paper focuses on scheduling tasks that deal with periodically occurring events. But in some situation, embedded systems may handle aperiodic events - events that occur just one time, or their period is too long to be considered as periodic. In this case, the system tries to minimize the response time of the task handling the aperiodic events [21]. Though we can handle the events by a polling server with a certain period, the polling server may cause unnecessary scheduling overheads and result in system utilization. Most of the existing algorithms are developed for preemptive scheduling, and few is available for non-preemptive fixed priority scheduling [22]. Thus the effective handling of aperiodic tasks in such an environment is also an important issue, and it will be our future work.

References

[1] F. Dabek, N. Zeldovich, M. Frans Kaashoek, D. Mazieres, and R. Morris. Event-driven programming for robust software. In Proceedings of the 10th ACM SIGOPS European Workshop, pages 186-189, September 2002.

[2] T. M. Parks and E. A. Lee. Non-preemptive real-time scheduling of dataflow systems. In Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing, pages 3225-3238, May 1995.

[3] K. Jeffay, D.F. Stanat, and C.U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In Proceedings of IEEE Real-Time Systems Symposium, pages 129-139, December 1991.

[4] L. George, N. Riviere, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor sche-

duling. Technical report, INRIA, 1996.
[5] M. Spuri. Analysis of deadline scheduled real-time systems. Technical report, INRIA, 1996.
[6] Y. Wang and M. Saksena, Scheduling fixed-priority tasks with preemption threshold. In Proceedings of IEEE International Conference on Real-Time Computing Systems and Applications, pages 328-335, 1999.
[7] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of ACM, 20(1):46-61, 1973.
[8] R. Gerber, S. Hong, and M. Saksena. Guaranteeing real-time requirements with resource-based calibration of periodic processes. IEEE Transactions on Software Engineering, 21(7):579-592, 1995.
[9] W. Wang, A. K. Mok, and G. Fohler. Pre-scheduling. Real-Time Systems, 30(1-2):83-103, 2005.
[10] F. Balarin and A. Sangiovanni-Vincentelli. Schedule validation for embedded reactive real-time systems. In Proceedings of the 34th Annual ACM/IEEE Conference on Design Automation, pages 52-57, 1997.
[11] L.-P. Chang. Event-driven scheduling for dynamic workload scaling in uniprocessor embedded systems. In Proceedings of the 2006 ACM Symposium on Applied Computing, pages 1462-1466, 2006
[12] P. Ramanathan and M. Hamdaoui. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. IEEE Transactions on Computers, 44(12):1443-1451, 1995.
[13] R. Wilhelm, E. Jakob, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem -- overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems, 7(3), Article No. 36, 2008.
[14] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. Real-Time Systems, 2(4):301-324, 1990.
[15] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised, Real-Time Systems 35(3):239-272, 2007.
[16] J.P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behaviour. In Proceedings of IEEE Real-Time Systems Symposium, pages 166-171, 1989.
[17] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. IEEE Transactions on Computers, 39(9):1175-1185, 1990.
[18] M. S. Branicky, S. M. Phillips, and W. Zhang. Scheduling and feedback co-design for networked control systems. In Proceedings of the 41st IEEE Conference on Decision and Control, pages 1211-1217, 2002.
[19] J.V. Busquets-Mataix, J.J. Serrano, R. Ors, P. Gil, and A. Wellings. Using harmonic task-sets to increase the schedulable utilization of cache-based preemptive real-time systems. In Proceedings of International Workshop on Real-Time Computing Systems Application, pages 195-202, 1996.
[20] T.-W. Kuo and A.K. Mok. Incremental reconfiguration and load adjustment in adaptive real time systems. IEEE Transactions on Computers, 48(12):1313-1324, 1997.
[21] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. IEEE Transactions on Computers, 44(1):73-91, 1995.
[22] W. Li, K. Kavi, and R. Akl. A non-preemptive scheduling algorithm for soft real-time systems. Computers and Electrical Engineering 33(1):12-29, 2007.

박 문 주
1996년 서울대학교 조선해양공학과(공학사). 1998년 서울대학교 컴퓨터공학과(공학석사). 2002년 서울대학교 전기컴퓨터공학부(공학박사). 2002년~2006년 LG전자 단말연구소. 2006년~2007년 IBM Ubiquitous Computing Lab. 2007년~현재 인천대학교 컴퓨터공학과 전임강사. 관심분야는 실시간시스템, 내장형시스템